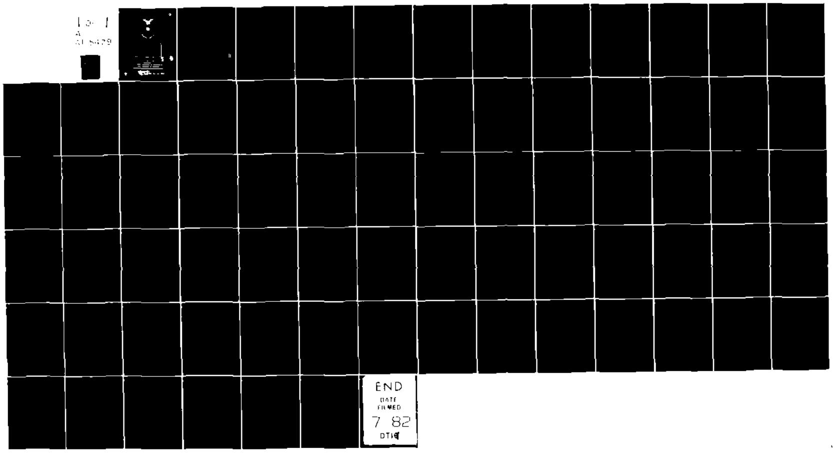
AD-A115 479   AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
TOWARD ADA: THE CONTINUING DEVELOPMENT OF AN ADA COMPILER.(U)
DEC 81 P R WERNER
UNCLASSIFIED AFIT/GCS/MA/81D-7   NL

1 of 1
A
115479



END
DATE
FILMED
7 82
DTIC

AD A115479

DTIC

S D

[illegible]

[illegible]

[illegible]

[illegible]

COPY

1

AFIT/GCS/MA/81D-7

TOWARD ADA: THE CONTINUING DEVELOPMENT OF AN ADA COMPILER

THESIS

AFIT/GCS/MA/81D-7 Patrick R. Werner
Capt USAF

AFIT/GCS/MA/81D-7

TOWARD ADA: CONTINUING DEVELOPMENT OF AN ADA COMPILER

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by

Patrick R. Werner, B.S.
Capt USAF

Graduate Computer Science

December 1981

PREFACE

I originally became interested in computers when I built a small analog computer as a high school science fair project. My interest in computers has continued to grow as I became more deeply involved with computers through my education and work.

My interest in languages and compilers grew with my increasing involvement with computers. Shortly before my assignment to the Air Force Institute of Technology, I received further motivation to find out what was involved in the makings of compilers and the structure of language. Several groups within the Air Force were attempting to determine which of the various dialects of JOVIAL the Air Force should establish as the standard. It was alledged that certain constructs of JOVIAL written in one manner would be easier to compile than another. I was at a disadvantage, not being able to fully weigh all the material which was presented.

The opportunity to take a sequence of courses in formal language and compiler construction as part of my graduate education was one I could not avoid. This lead to the renewal of my acquaintance with Ray Garlington. His initial efforts to build both a pseudo machine for Ada and an Ada compiler provided the basis for an excellent thesis project: the continuing development of the AFIT-Ada compiler.

The encouragement I received from Dr. Charles W. Roark, who taught the compiler sequence, and Roie R. Black, who became my thesis advisor, with Ray's constant enthusiasm, made the choice of any other thesis topic impossible.

The original focus of this thesis was to add type constructs to the minimal set of Ada constructs that Garlington had implemented in the bare Ada compiler. As I began to dig deeper into what was required in the compiler, and investigated what was already there, it became apparent that another major effort would be necessary to bring the AFIT-Ada compiler to the point were others could make use of the work of Garlington and myself.

As with any software development, particularly a continuing development by several individuals over a period of time, not only must it work correctly, but it must be written in such a way that it is understandable to each individual with a minimum of confusion. It must follow the standards that have been developed for good software engineering.

Garlington was primarily interested in the definition of the Ada pseudo machine. The construction of the Ada compiler was a secondary effort to verify the correct definition of the Ada pseudo machine. A rigorous effort to verify and validate the compiler was not made. Thus, it became a necessary part of this effort to correct errors in the code as they were discovered. In the end a new requirement surfaced, that of rewriting portions of the compiler into smaller, more comprehensible sections of code. For instance, the original compiler had a single case statement in the semantic analyzer that ran for over 30 pages of code.

I would like to thank Lt Col Raymond D. Bellum and the Air Force Avionics Laboratory for providing the DEC-10 System on which the research was conducted, the Carnegie-Mellon University SCRIBE document production system by which this thesis report was developed, and especially those fellow programmers and operators who provided assistance late at night, making this thesis possible.

I should also thank my advisor, Roie R. Black, for his continuing counsel and advice. Many thanks to my readers, Dr James P. Rutledge and Charles W. Richard, for their diligent efforts in helping me through this project. Dr Rutledge in particular provided numerous relevant articles and endless advice on Ada, both in person and through a continuing stream of messages relayed by the DEC-10 System.

Table of Contents

List of Figures

ABSTRACT

This thesis involved the continuing development of an Ada compiler, the AFIT-Ada compiler. Basic concepts of the compilation process were reviewed. The design and structure of the AFIT-Ada compiler were examined. Tests from the Ada Compiler Validation Capability system were modified for the subset of the Ada language currently implemented by the AFIT-Ada compiler. Those modified tests were run against the compiler to assure compliance with the Ada Reference Manual. Deviations from the language were corrected. Sections of the existing compiler were rewritten into a more structured manner. Recommendations were made for further development. This continuing development of the AFIT-Ada compiler was performed on the DEC-10 system of the Air Force Avionics Laboratory at Wright-Patterson AFB, Ohio.

1. INTRODUCTION

1.1 Objective

The objective of this thesis is to continue the development of an Ada compiler originally developed by Alan R. Garlington. This involved the analysis of the Ada language and the existing compiler. Originally, the initial area of the compiler to be improved was the type construct. However, during the initial study of the AFIT-Ada compiler it was discovered that assumptions as to the state and documentation of the existing compiler were not valid. Therefore, the revised objectives for this thesis were to review basic compilation concepts; provide further explanation of the design and structure of the AFIT-Ada compiler, restructure where necessary to provide more modular code, validate compliance with the Ada language, and to document the various steps in creating the compiler.

1.2 Background

Augusta Ada Byron, Countess Lovelace, the daughter of the poet Lord Byron, was a colleague of Charles Babbage and author of the first book on programming [Ref 22:647]. She provided formulae for solving problems on Charles Babbage's analytical engine. It is appropriate then, that the Department of Defense (DoD) has renamed DoD1 in her honor.

INTRODUCTION

DoD1, now Ada, is the outgrowth of an effort by the Department of Defense to reduce rising software costs which annually exceed three billion dollars ($3,000,000,000.00) [Ref 4:44]. One of DoD's most significant problems in the early 1970's was the proliferation of languages. Many new systems were acquired with custom languages and compilers - optimized for specific applications. The resulting languages were usually of poor quality. Many of the development projects failed because of language and compiler problems. There are many factors which contributed to these failures. Two of the most significant were time and independence of efforts. While experience has shown that it takes at least three years to achieve quality implementation of a new language, schedules often allocated only one year for language and compiler development. Attempts to design and code applications software in parallel with the language design and compiler implementation consistently met with unsatisfactory results [Ref 7:13].

The proliferation of programming languages had several other effects. Since each project was using a custom language, or special dialect of an existing language, in addition to the development of the language, compiler, and any support environment, the project director had only two sources for programmers to develop the application software: the relatively few individuals already familiar with the language, or the training of new programmers in that custom language. The programmers could not move readily from one project to another, and the software developed for one project could seldom be reused from one system to the next [Ref 5:11].

1.3 Assumptions

It was assumed that any continuing work on the development of an AFIT-Ada compiler would build upon the work accomplished by Garlington. This in turn predisposes several assumptions: an LR(1) grammar exists which provides the complete syntax for the Ada programming language; the existence of a program which will generate the parser tables from the LR(1) grammar; and that the LR grammar method, i.e. bottom-up parsing, is the desired approach. Since some of the preceeding terms may have rather specialized meanings, an explanation of these terms is provided in Chapter 2 and in Appendix E.

1.4 Overview

The current AFIT-Ada compiler implements only a subset of the Ada programming language. Included in this subset are integer variables, task and package declarations, functions and procedures, integer arithmetic or Boolean expressions, as well as the following statements: assignment, procedure, function or entry calls, exit, return, if, accept, and loop.

INTRODUCTION

The original compiler was implemented very quickly for such a large effort which lead to trade-offs in the coding. The overall goal of this effort is transform the existing code into a well structured software program, correct known errors, and test the compiler to verify proper adherence to the Ada language. Currently there is no validated Ada compiler available, so this continuing effort to develop an Ada compiler is a practical and meaningful project.

To be able to discuss the improvements to the AFIT-Ada compiler, requires an explanation of the concepts and structure of the AFIT-Ada compiler. However before this is done it is appropriate to review the basic concepts in constructing a compiler. Chapter 2 provides this examination of compilers and the compilation process. With this background, Chapter 3 provides an explanation of the AFIT-Ada compiler identifying how each of concepts examined applies to the AFIT-Ada compiler. Finally, Chapter 4 is devoted to an explanation of the improvements made to the original AFIT-Ada compiler.

Appendices are included which provide a Compiler User's Guide, an explanation of the steps necessary to create the compiler, examples of the tests used to validate the compiler, and a brief list of important terminology.

2. COMPILERS

2.1 Compilers

Translators are programs that accept an input program written in one language, the source language, and transform that program into another output language, the target language. Compilers are programs that accept programs written in a high-level language, in this case Ada, and translate them to an equivalent representation in a machine language. Compilers map the source language into the target language. The mapping is accomplished in either a single pass or multiple passes. Multipass compilers map the source language by degrees into the target language [Ref 16:2].

Another view is that the term translator refers to a set which contains such things as compilers, assemblers, and interpreters [Ref 19:41]. Compilers translate the high-level language statement into as many low-level language statements as necessary to carry out the required operations while retaining the meaning of the original statement. Compilers produce either assembler code, which must then be assembled, or machine code, which may be executed directly to perform the computation desired by the original statement in the source language. Assemblers perform the comparatively simple translation from the assembly, or mnemonic, code to the machine, or object, code. Interpreters translate each high level language statement into executable code, executing the code directly as the translation is accomplished, without producing an object code file as output.

2.2 Compilation

It is useful to subdivide the process of translating a source program to a program in the target language into a sequence of simpler subprocesses called phases [Ref 2:10]. A simple compiler model might consist of the following phases:

- lexical analysis
- syntax analysis
- semantic analysis
- code generation
- interpretation

The meaning of each of these five phases is examined below.

2.2.1 Lexical Analysis

The first phase of any compiler is the lexical analyzer. The lexical analyzer scans and screens the input stream. Scanning involves finding substrings of characters that constitute textual elements. These textual elements are the words, punctuation, single- and multi-character operators, comments, spaces, and maybe line boundary characters. Screening involves the discarding of some textual elements, such as spaces and comments, and the recognition of reserved symbols used in the particular language [Ref 14:109]. The lexical analyzer transforms the input stream of characters into a stream of tokens. Tokens are the most primitive, meaningful element of a language from which the sentences of a language are composed. For example, the word is the most primitive element of the English language. Thus, CAT, DOG, and SEE are tokens, even though they themselves are constructed of letters. The letters, by themselves, having no meaning, are not recognized independently in a sentence, but only as part of a word.

2.2.2 Syntactic Analysis

The second phase is the syntax analyzer, or parser. The parser groups the tokens into syntactic structures such as declarations, statements and control structures. The parser determines if a particular stream of tokens is formed according to the rules, or grammar, of the particular language. The theory of grammars, and in particular LR(k) grammars, is the basis for successful automation of syntax analysis [Ref 2:11].

The development of the theory of grammars has immensely improved the process of compiler writing. For example, the first FORTRAN compiler took 18 man-years to implement. Today, however, compilers can be built with much less effort [Ref 1:1]. One of the reasons for the compression of the time required for the development of compilers is the change of philosophy of compiler writers. In the past each compiler was written as an independent entity, beginning from scratch. Today, there are compiler-compilers, or translator writing systems, which take a description of the language as input, and produce a parser as output. Most frequently the syntactic specification of the language is presented in yet another formal language called Backus normal form (BNF). Thus lexical and syntactic routines analysis may be generated automatically. The next phase of the compilation process, the semantic analysis, has yet to benefit as completely from automation.

2.2.3 Semantic Analysis

While the syntax analyzer assures that the program is well-formed, semantic analysis assures that the program is meaningful. The semantic analyzer determines the type of intermediate results, checks that arguments are of the legal type and number for an application of an operator, and determines the operation denoted by the operator. For example, + could denote fixed or floating addition, or possibly the logical OR operation [Ref 1:20]. Errors which were not detectable by lexical or syntactic analysis, such as the use of undeclared variables, are identified in the semantic analysis [Ref 2:11].

As an example, if the working language is English then the following sentence can be seen to have problems:

THE CHAIR ATE T CAT

In scanning the sentence, the letters are distinguished from the blank page and grouped into words, which corresponds to the lexical analysis. The single letter T is detected as an error during the syntactic analysis, since there is no word, or token, corresponding to T in the English language. Finally, if the T was replaced by THE, most observers would probably still reject the sentence, even though it is now syntacticly constructed in accordance with the grammatical rules of English. While properly formed of nouns, verbs, and articles, the sentence does not make sense. It is not meaningful. The noun CHAIR may not be the subject of the verb ATE. However, there might be disagreement as to whether the sentence

THE CAT ATE THE CHAIR

was meaningful. Semanticly correct, since CAT may be the subject of the verb ATE, at execution an error could be detected depending if the chair was constructed of chocolate or oak.

The semantic analysis is an area that has yet to be extensively automated, primarily due to the difficulty in identifying the semantics of the language in a rigid, formal specification. The syntax can be specified using BNF, but the semantics are usually explained in text. Ada is attempting to alleviate this difficulty by developing a formal specification of the semantics for Ada through the use of DIANA, a Descriptive Intermediate Attributed Notation for Ada [Ref 15].

2.2.4 Code Generation

Once the source program is scanned by the lexical analyzer to isolate the elements of the language; parsed by the syntax analyzer to verify grammatical correctness; and synthesized by the semantic analyzer, checking to assure it is meaningful; the code to perform the required actions must be generated. Actually, several types of code may be generated. Frequently, intermediate code is generated which is then optimized. This intermediate language program can be a sequence of quadruples or triples, a tree, or a postfix Polish string [Ref 1:518]. The raw object code generated by the code generator may be of several different kinds, depending on the purpose of the compiler: it may be machine code for some particular target machine, or it may be a special intermediate file designed to be further processed by a loader. It may also be a specially designed intermediate code that must be further translated by another system into machine code or a loader structure [Ref 3:9-10].

2.2.5 Interpretation

While the compilation process is completed, in the formal sense, with the code generation, to be useful a linker/loader or interpreter is often included as part of the compiler. Thus, the final phase in this model of a compiler is the interpretation. Whatever structures, datum, and operations have been derived from the source program which was initially input to the compiler, the last step is to execute the code produced during the code generation. Often the code produced during the code generation may not be the machine code for the target machine, in which case the code must be interpretted such that actual machine operations are possible.

2.3 Grammars

The techniques employed in modern compilers are based on the study of the concept of grammars. It is appropriate therefore to review the various aspects of a grammar. A grammar is a set of structural rules defining the legal context of tokens in sentences [Ref 3:15]. A grammar provides a precise, yet easy to understand, syntactic specification for the programs of a particular programming language. Grammars are composed of four quantities: terminals, non-terminals, a start symbol, and productions. Terminals are the basic strings of a language. Nonterminals are special symbols that denote sets of strings. The start symbol is a special nonterminal which denotes the language [Ref 1:127]. Productions are the grammatical rules which determine how new forms may be generated or produced [Ref 26:282].

To develop a clearer understanding of these terms, examine the following simple grammar for an English sentence:

```
 1. <Sentence>   ::=  <Subject> <Predicate>
 2. <Subject>    ::= <Noun> | <Article> <Noun>
 4. <Predicate>  ::= <Verb> <Object>
 5. <Object>     ::= <Noun> | <Article> <Noun>
 7. <Noun>       ::= CAT | CHAIR | MOUSE
10. <Verb>       ::= ATE | SAW
12. <Article>    ::= THE | A
```

Figure 2-1: Example Grammar

In this grammar the terminals are the words CAT, CHAIR, MOUSE, ATE, SAW, THE, and A. The nonterminals, enclosed in <> pairs, are Sentence, Subject, Predicate, Object, Noun, Verb, and Article. The start symbol is Sentence. Figure 2-1 represents the production rules for the grammar. Notice that there may be more than one production for a nonterminal, one production is separated from an alternative production by the vertical bar |. In this grammar <Subject> has two productions, production 2 reducing <Subject> to <Noun> and production 3 reducing <Subject> to <Article> <Noun>.

2.4 Methods

Various methods have been developed to systematize the generation of compilers. The three concepts of primary interest are top-down parsing, bottom-up parsing, and syntax-directed translation.

As their names imply, bottom-up parsers build parse trees from the bottom, or leaves, to the top, or root, while top-down parsers start with the root and work down to the leaves. In both cases the input to the parser is being scanned from left to right, one symbol at a time.

The bottom-up parsing method is called "shift-reduce" parsing because it consists of shifting input symbols onto a stack until the right side of a production appears on the top of the stack. The right side may then be replaced by, or reduced to, the symbol on the left side of the production and the process is repeated until the system goal symbol appears on the top of the stack [Ref 1:146]. The bottom-up parsing methods are frequently used on languages represented by LR grammars.

There are two types of top-down parser. The general form of top-down parsing may involve backtracking which requires repeated scans of the input [Ref 1:174]. The recursive-descent parser eliminates the need for repeated scans of the input. This is accomplished by employing either recursive procedures or a tabularized implementation, predictive parsing, to detect the proper alternative by looking at the first symbol it derives [Ref 1:180]. Top-down parsing methods often are employed on LL grammar represented languages.

Syntax-directed translation is a technique in which the actions of the syntax analysis phase guide the translation [Ref 1:16-17]. Of particular interest is the translation from token streams to intermediate code. There are two basic issues of interest: what intermediate code to generate for each programming language construct and an algorithm to implement the code generation. A syntax-directed translation is a context-free grammar in which a program fragment called an output action, or a semantic action or semantic rule, is associated with each production. In a bottom-up parser, the action is taken when the right hand side of a production is reduced to the left hand side of the production. The semantic action might involve the computation of values for variables belonging to the compiler, the generation of intermediate code, the printing of error diagnostics, or the placement of some value in a table [Ref 1:246].

3. AFIT-Ada COMPILER

Now that the basic concepts of a compiler have been discussed, the specifics of the AFIT-Ada Compiler may be presented. But first it is appropriate to identify its many sources.

3.1 Origin

The AFIT-Ada compiler was created from a multiplicity of sources. The primary source was the Reference Manual for the Ada Programming Language [Ref 21]. From this definition of the language Intermetrics translated the syntactic structures of Ada into a BNF like grammar [Ref 18]. This grammar was in turn an input to the Lawrence Livermore Laboratory LR Automatic Parser Generator [Ref 24], which produced the parser tables. Wirth's PL/0 Compiler, [Ref 26:307-347], provided the structure and much of the code for the AFIT-Ada Compiler Parser. Finally, Garlington's architecture for the Pseudo-Ada Machine came from Barrett and Couch's AOC machine, [Ref 3:377-463], and Wirth's hypothetical PL/0 processor, [Ref 26:331-333]. For an explanation of how the compiler is constructed from all of these various pieces see CREATING THE COMPILER, Appendix C.

3.2 General Approach

Some compilers start at the begining of the compiler model discussed earlier with the scanner, proceeding through the various steps outlined in Chapter 2 in a sequential, but continuous, fashion. The AFIT-Ada compiler applies the bottom-up approach throughout. It is a finite state automaton which processes the input stream of symbols, performing transitions and reductions as needed until the system goal symbol is obtained.

```
      (********************)
      (*                  *)
      (*   MAIN PROGRAM   *)
      (*                  *)
      (********************)

    BEGIN  (*  ADA COMPILER  *)

      INITIALIZE;

      PARSE;

    99:

      IF  LINERRPTR <> 0            THEN  PRINTLINERRORS ;
      IF  PROGRAMERRFLAG            THEN  PRINTERRORS    ;
      IF  SWITCH [PRINTCODE]        THEN  GOPRINTCODE    ;
      IF  SWITCH [PRINTTABLE]       THEN  GOPRINTTABLE   ;
      IF  (SWITCH [EXECUTE]) AND
          (NOT  PROGRAMERRFLAG )    THEN  INTERPRET      ;

      TERMINATION

    END.
```

Figure 3-1: Compiler Main Program

3.3 Main Procedure

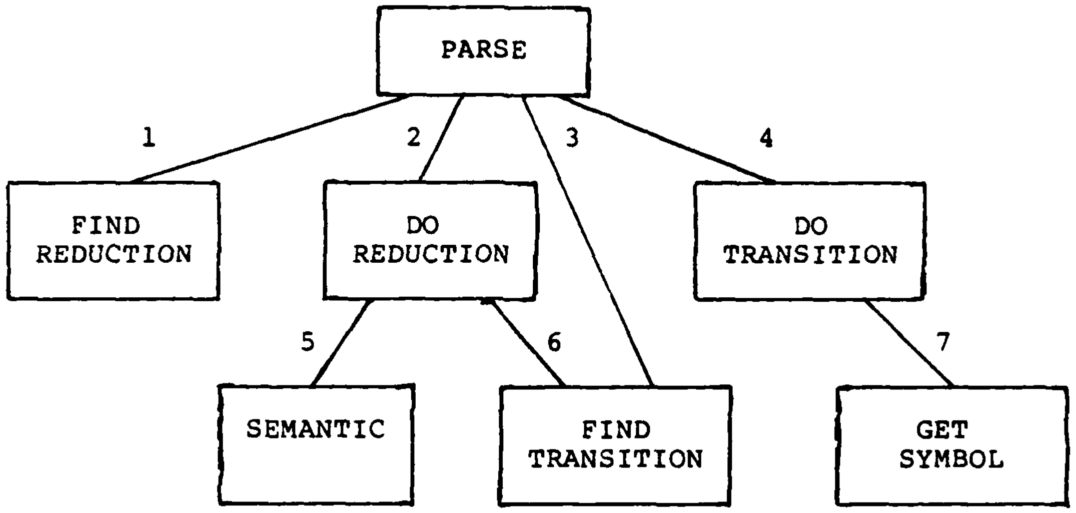
The main program of the AFIT-Ada Compiler is quite simple, as Figure 3-1 indicates. First, initialization is accomplished, then parsing is performed. At the conclusion of the parsing various print routines for the errors, code, and symbol table are executed if the the appropriate switch or flag so indicates. If there are no program errors and execution was requested, the code generated by the compiler is interpreted. Finally, a termination routine is invoked.

```
REPEAT

  IF a reduction is possible while in the current
     state with the current look-ahead token

     THEN do the reduction

     ELSE IF a transition is possible while in the current
             state with the current look-ahead token
             THEN do the transition

     ELSE there is a syntax error in the source language input

UNTIL the current state is the final state
```

Figure 3-2: Parsing Algorithm

3.4 Parse

As stated above, the compiler is a finite state automaton which accepts the input stream of symbols, performing reductions and transitions as required until the system goal symbol is obtained. The parser follows the algorithm presented in figure 3-2. The major structure of the compiler, the parser, is present in Figure 3-3 from Garlington's Thesis [Ref 10:55].



	IN	OUT
1	current state and token	production #
2	production #, stkptr	new state #, stkptr
3	current state and token	new state #
4	new state #, stkptr	stkptr, current state # token description
5	production #, stkptr	stkptr
6	stack[stkptr].state, left-hand side production #	new state #
7		token description

Figure 3-3: Parser Structure Chart

The module at the top of the diagram, PARSE, calls four subordinate modules named FINDREDUCTION, DOREDUCTION, FINDTRANSITION, and DOTRANSITION. When PARSE invokes FINDREDUCTION, it provides module FINDREDUCTION with the information required to describe the current state and the current look-ahead token. Upon return, FINDREDUCTION provides PARSE with the production number [Ref 10:54]. Whereupon PARSE passes the production number and current semantic stack pointer to module DOREDUCTION. DOREDUCTION passes the production number and current semantic stack pointer to SEMANTIC. SEMANTIC generates the object sentence for the source language sentence, reduces the semantic stack as appropriate, and passes control and possibly a new semantic stack pointer back to DOREDUCTION. DOREDUCTION passes the state from the semantic stack and the production number to FINDTRANSITION which determines the new state of the automaton, passing control back through DOREDUCTION to PARSE. PARSE may now pass the current state and production number to FINDTRANSITION receiving back the new state. The new state and semantic pointer are passed to DOTRANSITION which stacks the current look-ahead symbol, the look-ahead symbol's attributes, and the parser's next state. Then DOTRANSITION invokes GETSYM which obtains the next token and its attributes from the input stream.

This LR(1) parser is an implementation of the bottom-up parsing method discussed in Chapter 2. The parsing program has the property of immediately pursuing a new subgoal whenever one appears. This implies two major requirements. The first is that some mechanism will retain any necessary information while the program is pursuing the new subgoal. The second is that the syntax must be void of choices between several alternative non-terminal elements [Ref 26:298].

3.5 Lexical Analysis

Obtaining the next symbol from the input stream is part of the lexical analysis. The lexical analysis is accomplished within the major procedure GETSYM. The lexical analyzer screens and scans the input stream of symbols, returning the token along with any attributes with which they might be associated. The lexical analyzer does this by building a descriptor record for the token. The token descriptor contains the symbol name, the integer, real, or character value, and a pointer to the token's symbol table entry for the ASCII representation.

3.6 Semantic Stack

Having built a descriptor record for the token the lexical analyzer has completed its function. Thus control is passed back from GETSYM to DOTRANSITION. Here the semantic stack is constructed. The semantic stack is the mechanism which will retain information as new subgoals are pursued. The semantic stack stores the current state, the current token, the description of the token constructed by the lexical analyzer, additional information such as the type of expression and a couple of utility variables.

3.7 Semantic Actions

While DOTRANSITION continually adds to the semantic stack whenever it is invoked, DOREDUCTION has the opposite effect. DOREDUCTION is the procedure which implements the syntax-directed translation scheme discussed in Chapter 2. Passed the production number and a pointer to the semantic stack, DOREDUCTION invokes the appropriate semantic routine to accomplish the necessary semantic action by passing SEMANTIC the production number. Once the semantic action is accomplished, the semantic stack is decremented by the length of the production which determined which semantic routine to apply.

3.8 Symbol Table

A symbol table is the compiler structure which associates identifiers with their attributes. A declaration is a statement which assigns attributes to some identifier. An identifier is declared when it has appeared in a declaration. An identifier is said to be referenced or used in a statement in which it appears, but in which no attributes are added to the identifier's attribute set [Ref 3:322].

3.8.1 Concerns

Two important concerns with regard to identifiers are their visibility and scope. Visibility is where the identifier is known on its own. Scope is the area where the identifier is defined and determines where an identifier can be made visible. Name visibility is enforced with a compile time environment stack, stacking rules and special symbol table access routines.

An entry in the environment stack contains information on the name of the environment, whether or not the environment acts as a package visible part, and whether or not the environment is directly visible. Initially, the environment stack is empty [Ref 10:59].

4. IMPROVEMENTS

Now that the concepts which are important to the compilation process have been explained and the AFIT-Ada compiler has been examined, the efforts involved in this thesis may be identified. The work done to continue the development of the AFIT-Ada compiler fell into two categories: improvement of the code and validation that the compiler correctly implements the subset of the Ada language.

4.1 Code Improvements

There were two areas where the code required improvements before modifications could be made to expand the implemented Ada subset. The first was the formatting of the code. As originally written, the code was difficult to read. Sections of code and comments were intermixed, several executable statements were included on single lines of code, there was little vertical spacing separating sections of code, and sections of the code were lengthy and involved. The last of these lead to the second area which required improvement; restructuring the code in a more modular fashion to improve not only its readability, but to simplify understanding and modification of the code.

An example of how reformatting and restructuring the code can improve the understanding of the code can be seen in the lexical analysis of the input stream. Appendix A provides the original code for the lexical analysis of a possible number. Some sections are redundant, flags are passed back and forth to control execution, comments and code are not always easily distinguished, and there is even an error since there is no provision for the transliteration of the character ':' for the character '#' in a based number.

The current lexical analyzer uses a modular approach, replacing the original four page case statement with a short case statement which calls an appropriate procedure for various situations of lexical interest. The procedure LEXNUMBER which performs the lexical analysis of a possible number, likewise employs a modular approach. LEXNUMBER calls the procedure for a integer digit, LEXDIGITI until a character is encountered which terminates the string of digits whereupon the appropriate procedure for a based number, a real number, or an exponent is invoked.

One of the additional tools added during this thesis project was the PRINTSTACK toggle. For an explanation of toggles see Appendix B. PRINTSTACK is a switch which allows the compiler developer to print out the semantic stack. This is useful to assure that the semantic actions which are implemented are removing the correct tokens, descriptors and attributes from the semantic stack. While the semantic routines are processing a reduction or transition they often refer to the symbol table to determine if an string of characters is defined in some particular fashion.

4.2 Need for Validation

The Ada programming language is a very powerful high order language. The features of the language which make it powerful also make the language rather involved and sophisticated. This in turn means that the implementation of the language is far from trivial. Currently there are 93 terminals, 216 non-terminals, 444 productions, and 751 possible states in the LR(1) grammar used to represent the Ada language. Thus it was not expected that a complete and fully accurate compiler would be produced during the first attempt. That the original AFIT-Ada compiler was not free from bugs has been confirmed.

4.3 Validation

Compiler validation tests are intended to show whether a compiler correctly implements a particular programming language. The Ada Compiler Validation Capability (ACVC) tests were developed for DOD use to enforce (and encourage) correct implementation of Ada [Ref 13:57].

4.4 Specific Problems

While running several programs through the original AFIT-Ada compiler several discrepancies in the handling of the source programs were discovered. One of the problems which caused the most concern was the restricted set of characters which the AFIT-Ada compiler would handle. The AFIT-Ada compiler only processed the characters of the restricted character set. Any character other than upper-case letters, numerals, and certain special characters ('-', '.', '&', etc.) were encountered the compiler would remain in an infinite loop. When a test case was applied to the compiler which contained lower case letters and the compiler took an inordinate amount of time without finishing the compilation, this investigator began to question how fast or efficient an Ada compiler could be if a compiler which implements a very restricted subset of the Ada language took as long as the AFIT-Ada compiler appeared to be taking. Thus, it was a great relief to discover the error and find that the compiler was able to compile the test programs very quickly.

Of course this was not the only error discovered. Others, such as the inability to distinguish between reserved words and attributes, were equally unsettling. Thus it became apparent that it was advisable to assure the proper functioning of that subset of the Ada language which had been implemented before extending the compiler's capabilities.

4.5 Ada Compiler Validation Capability

The test programs used to validate the AFIT Ada compiler were obtained from Dr John B. Goodenough via the ARPAnet host at USC-ECLB. Since it was known that the AFIT-Ada compiler was attempting to function on only a subset of the Ada language, the programs obtained from Dr Goodenough required modification prior to use to bring them into conformance with the currently implemented Ada constructs.

4.5.1 Classes of Tests

The ACVC tests are classified according to the general nature of their criteria for passage or failure. There are six classes of test [Ref 13:60]:

Class A -	tests are passed if no errors are detected at compile time.
Class B -	tests are illegal programs. Tests pass if all the errors they contain are detected at compile time and no legal statements are considered illegal by the compiler.
Class C -	tests consist of executable self-checking programs. They are passed if they complete execution and do not report failure.
Class D -	tests are capacity tests.
Class E -	tests determine how an implementation has interpreted ambiguity in the language.
Class L -	tests consist of illegal programs whose errors are not detected until link time.

Examples of test runs for the ACVC tests, as well as a few others, are provided in Appendix D.

IMPROVEMENTS

This discussion of the importance of validation to a significant software effort brings to a conclusion the major portion of this thesis report. However, the development of the AFIT-Ada compiler accomplished with this thesis effort, like any scientific investigation, does not preclude future contributions. Therefore, the conclusion of this report is a set of recommendations for further development of the AFIT-Ada compiler.

5. RECOMMENDATIONS

Not too long ago, computer programmers were stereotyped as rather strange and eccentric people. The programs that they wrote were considered some sort of mystical communication not to be understood by others. Such attitudes are, hopefully, a thing of the past. Today an entire body of science of software engineering is developing. While the development of the AFIT-Ada compiler is a student effort, that does not reduce any of the well documented problems of software development and maintenance. The primary consideration, if the development of an Ada compiler will be a continuing effort, is to establish a method of configuration control for the various versions of the compiler.

Coupled with the establishment of some configuration control methods should be the specification of how various projects working from the same base will interface, both during and after the development as thesis projects. A simple example of the type problem which could develop is the addition of pragmas or pragma options to the compiler. Pragmas are the Ada construct which permit the source program to provide information to the compiler. The AFIT-Ada compiler has a pragma TOGGLE which permits the user to control compiler options during the translation of the source program. For a more complete explanation of the TOGGLE pragma see Appendix B, which begins on page 40. During the development of an interactive debugger for the AFIT-Ada compiler, a toggle DEBUG was added to the compiler to control compiler actions required for debugging. During the continuing development of the AFIT-Ada compiler, a toggle PRINTSTACK was included to permit a printout of the semantic stack which is useful when attempting to debug the compiler itself. Had there not been a close working relationship between the implementers of these two projects, a conflict could have developed which would preclude the merging of these two versions of the compiler.

As was identified earlier, the objective which was originally intended to be the focus of this thesis effort was the addition of the type construct. Since other issues became more critical to the continuing development of the AFIT-Ada compiler, the addition of type structures remain as a valid requirement for a compiler which implements the Ada language. The compiler now is in a state of development where it is reasonable to provide such an extension.

Once the type construct is included, further development of the AFIT-Ada compiler could, and should, be done in the Ada language itself. Such bootstrapping of the compiler would require a version of the minimal compiler to be written in Ada. Since the current version of the AFIT-Ada compiler is a large program, it would be useful to have a PASCAL to Ada translator. In addition to aiding the bootstrapping of the compiler, this would prove useful for the translation of many standard or library PASCAL programs which could be useful for Ada programmers. Thus the development of a PASCAL to Ada translator would be an excellent area for future work.

The use of the LR Parser Generator, see Appendix C which begins on page 51, along with a syntax-directed translation scheme minimizes the effects which changes in the Ada grammar could cause in the compiler. However, the current use of the production number to identify the semantic action to be implemented implies the semantic code would be changed with each change to the grammar. Thus one should consider using a table driven method for the semantic routines to preclude the need to rewrite the semantic case statement if the Ada grammar provided to the LRS parser generator is changed and the production numbers, states (configuration sets), etc. change.

Currently, the use of the interpreter to perform the execution of the Ada program requires the source program to be compiled again for each execution. It could be desirable to separate the interpretation from the compilation so that the program can be executed several times without recompiling. This would require the Ada pseudo code to be written to an output file.

At present the only way to control the actions of the compiler is through the TOGGLE pragma. This requires that the source code be modified if the selection of different compiler options is desired. It could prove helpful to make toggles controllable from the system level, so that the test source program could be rerun with various options without altering the source code.

As currently implemented, the TRACEPARSE toggle produces a listing of the productions and transitions which is rather lengthy. See Appendix B, beginning on page 40, for an explanation of the TRACEPARSE toggle and Figure B-6, page 50, for an example printout of a program which used the TRACEPARSE toggle. A useful addition to the tools available to the user would be to write an output routine which maps the TRACEPARSE output into a listing which includes the actual productions and transitions from the LR Parser Generator rather than just the production and transition numbers. This would provide a much more readable trace for use when tracing the parse of an Ada program.

BIBLIOGRAPHY

1. Aho, Alfred V. and Jeffrey D. Ullman. Principles of Compiler Design. Addison-Wesley Publishing Company, Reading, Massachusetts, 1979.

2. Aho, Alfred V. Translator Writing System: Where Do They Now Stand?. Computer 13, 8 (August 1980), 9-14.

3. Barrett, William A. and John D. Couch. Compiler Construction: Theory and Practice. Science Research Associates, 1979.

4. Boehm, Barry W. Software Engineering: R&D Trends and Defense Needs. In Peter Wegner, Ed., Research Directions in Software Technology, The MIT Press, Cambridge, Massachusetts, 1980, pp. 44-86.

5. Braun, Christine L. Ada: Programming in the 80's. Computer 14, 6 (June 1981), 11-12.

6. Brozovic, Richard L. JOVIAL(J73) to Ada Translator System. Master Th., Air Force Institute of Technology, December 1980.

7. Carlson, William E. Ada: A Promising Beginning. Computer 14, 6 (June 1981), 13-15.

8. DeRemer, F., T. Pennello, and W. M. McKeeman. Ada Syntax Chart. SIGPLAN NOTICES 16, 9 (September 1981), 48-59.

9. Dewer, Robert R. K., et al. The NYU Ada Translator and Interpreter. Proceedings, Computer Software and Applications Conference, New York, October, 1980, pp. 59-65.

10. Garlington, Alan R. Preliminary Design and Implementation of an Ada Pseudo-Machine. Master Th., Air Force Institute of Technology, March 1981.

11. Glass, Robert L. Software Soliloquies. Computing Trends, Seattle, 1981.

12. Goodenough, John B. Ada Compiler Validation Implementers' Guide. TR 1067-2.3 AD-A091760, SofTech, Inc., October, 1980. DARPA/IPTO

13. Goodenough, John B. The Ada Compiler Validation Capability. Computer 14, 6 (June 1981), 57-64.

14. Goos, G., and J. Hartmanis, editors. Lecture Notes in Computer Science. Volume 21: Compiler Construction. Springer-Verlag, New York, 1976.

15. Goos, G. and Wm. A Wulf, editors. Diana Reference Manual. Carnegie-Mellon University and Univeritaet Karlsruhe, March, 1981.

16. Hartmann, Alfred C. Lecture Notes in Computer Science. Volume 50: A Concurrent Pascal Compiler for Minicomputers. Springer-Verlag, New York, 1977.

17. Ichbiah, J. D. et al. Rationale for the Design of the ADA Programming Language. SIGPLAN NOTICES 14, 6 (June 1979), 1-1,15-12.

18. Spector, David. Intermetrics Ada Grammar. Private communication to Dr Rutledge via an ARPANET message on 6 Oct 80.

19. Steele, Dennis R. An Introduction to Elementary Computer and Compiler Design. North-Holland, New York, 1978.

20. Stein, Jess, ed. The Random House Dictionary of the English Language. Random House, New York, 1967.

21. , United States Department of Defense. Reference Manual for the Ada Programming Language, July, 1980.

22. Wegner, Peter. A self-assessment procedure dealing with the programming language Ada. Communications of the ACM 24, 10 (October 1981), 647-677.

23. Weingarten, Frederick W. Translation of Computer Languages. Holden-Day, San Francisco, 1973.

24. Wetherell, Charles and Alfred Shannon. LR Automatic Parser Generator and LR(1) Parser. Livermore, California: Lawrence Livermore Laboratory, June 1979

25. Wirth, N. The Design of a PASCAL Compiler. Software--Practice and Experience 1, 4 (1971), 309-333.

26. Wirth, Nicklaus. Algorithms + Data Structures = Programs. Prentice-Hall, Englewood Cliffs, N.J., 1976.

A. COMPILER CODE

A.1 Introduction

Several small sections of the AFIT-Ada compiler are included. Examples of the compiler code as originally written by Garlington and the code produced as part of this thesis effort are displayed.

A.2 Sample Code

```
  IF CH = 'E' THEN BEGIN  (*EXPONENT PART*)
    DECIMALFLAG := FALSE; DES.CHARVAL := 'R';
    IF SCALE = 0 THEN DES.REALVAL := DES.INTVAL;
    GETEXPONENT (SIGN,EXPONENT);
    IF SIGN THEN
      SCALE := SCALE - EXPONENT
    ELSE
      SCALE := SCALE + EXPONENT;
    END;  (*THEN EXPONENT_PART*)
  IF SCALE <> 0 THEN BEGIN  (*ADJUST SCALE*)
    R := 1;  SIGN := SCALE < 0;
    SCALE := ABS (SCALE);  FAC := BASE;
    REPEAT
      IF ODD (SCALE) THEN R := R * FAC;
      FAC := SQR (FAC);
      SCALE := SCALE DIV 2
    UNTIL SCALE = 0;
    IF SIGN THEN
      DES.REALVAL := DES.REALVAL / R
    ELSE
      DES.REALVAL := DES.REALVAL * R;
    END;  (*THEN ADJUST_SCALE*)
END;  (*CASE OF '0'..'9'*)
```

Figure A-2: Original Number Lexical Analyzer (cont)

```
'0','1','2','3','4','5','6','7','8','9' : BEGIN
(*DES.CHARVAL IS USED TO SIGNAL WHETHER THE NUMBER IS
  REAL (CHARVAL = 'R') OR  INTEGER (CHARVAL NOT
  CHANGED FROM INITIALIZED VALUE  *)
  NEXTSYM := NUMTOK;  BASE := 10;
  SCALE := 0; DECIMALFLAG := FALSE; BASEDNUMBER := FALSE;
  GETNUMBER (CH,BASE,DECIMALFLAG,NUMDIGITS,DES.INTVAL,DES.REALVAL);
  IF CH = '#' THEN BEGIN    (*BASED NUMBER*)
    BASEDNUMBER := TRUE;
    BASE := DES.INTVAL;
    IF BASE > 16 THEN BEGIN  (*BASE ERROR*)
      ERROR (6,1,-1);
      BASE := 16
      END;  (*BASE ERROR*)
    INCHAR (CH);
    GETNUMBER (CH,BASE,DECIMALFLAG,NUMDIGITS,DES.INTVAL,DES.REALVAL);
    IF CH = '.' THEN BEGIN  (*BASED 'DECIMAL' PART*)
      DECIMALFLAG := TRUE;
      DES.REALVAL := DES.INTVAL;  DES.INTVAL := 0;
      DES.CHARVAL := 'R';
      INCHAR (CH);
      GETNUMBER(CH,BASE,DECIMALFLAG,NUMDIGITS,DES.INTVAL,DES.REALVAL);
      SCALE := - NUMDIGITS;
      END;  (*THEN*)
    IF CH <> '#' THEN ERROR (3,1,0);  INCHAR (CH);
    END;  (*THEN BASED_NUMBER*)
  IF CH = '.' THEN BEGIN    (*DECIMAL PART*)
    INCHAR (CH);
    IF CH = '.' THEN
      DOTDOTFLAG := TRUE    (*NEXTSYM WILL BE DOTDOTTOK*)
    ELSE BEGIN
      IF BASEDNUMBER THEN ERROR (3,1,-1);
      DES.REALVAL := DES.INTVAL;
      DES.INTVAL := 0; DECIMALFLAG := TRUE;
      DES.CHARVAL := 'R';
      GETNUMBER(CH,BASE,DECIMALFLAG,NUMDIGITS,DES.INTVAL,DES.REALVAL);
      SCALE := - NUMDIGITS;
      END;  (*ELSE*)
    END;  (*THEN DECIMAL_PART*)
```

Figure A-1: Original Number Lexical Analyzer

```
CASE CH OF

'A','B','C','D','E','F','G','H','I',
'J','K','L','M','N','O','P','Q','R',
'S','T','U','V','W','X','Y','Z'           :  LEXLETTER        ;

'0','1','2','3','4','5','6','7','8','9' :  LEXNUMBER        ;

'|','&','(',')','+',',',';','!'   :        LEXSINGLESYMBOL ;

'''' :                                     LEXQUOTE         ;

'-'  :                                     LEXDASH          ;

'"'  :                                     LEXDOUBLEQUOTE   ;

'='  :                                     LEXEQUAL         ;

'.'  :                                     LEXDOT           ;

'*'  :                                     LEXSTAR          ;

':'  :                                     LEXCOLON         ;

'/'  :                                     LEXSLANT         ;

'<'  :                                     LEXLESSTHAN      ;

'>'  :                                     LEXGREATERTHAN   ;

'®','$','#','[',']','%','_','?','@','©' :  INVALID          ;

OTHERS :  INVALID               (* - PREVENTS INFINITE LOOP - *)
                                (* HANDLES CHARACTERS NOT     *)
                                (* RECOGNIZED BY THE COMPILER *)
                                (* - LIKE TABS                *)

END  (* CASE OF CH *)
```

Figure A-3: Revised Lexical Analyzer

```
(*************************************)
(*                                   *)
(*   INTERNAL PROCEDURE LEX_NUMBER   *)
(*                                   *)
(*************************************)

PROCEDURE LEXNUMBER ;

BEGIN

  NEXTSYM     := NUMTOK ;
  BASE        := 10 ;
  NUMDIGITS   :=  0 ;
  SCALE       :=  0 ;

  DIGITI( NUMDIGITS, DES.INTVAL ) ;

  CASE  CH OF

    '#'       :  BEGIN
                   BASEDNUMBER ;
                   IF  CH <> '#'  THEN  ERROR( 3, 1, 0 ) ;
                   INCHAR( CH ) ;
                 END ;

    ':'       :  BEGIN
                   BASEDNUMBER ;
                   IF  CH <> ':'  THEN  ERROR( 3, 1, 0 ) ;
                   INCHAR( CH ) ;
                 END ;

    '.'       :  REALNUMBER ;

    'E','e'   :  GETEXPONENT

  END ; (* CASE CH OF *)

  IF  SCALE <> 0  THEN  ADJUSTSCALE

END ;  (*  LEX_NUMBER  *)
```

Figure A-4: Revised Number Lexical Analyzer

```
(*****************************************)
(*                                       *)
(*   INTERNAL PROCEDURE DIGIT_INTEGER    *)
(*                                       *)
(*****************************************)

PROCEDURE DIGITI ( VAR DIGITS, VALUE : INTEGER ) ;

(************************************************)
(*   PROCEDURE DIGITI IS INTERNAL TO GETSYM     *)
(*   PASSED THE CURRENT VALUE OF THE INTEGER    *)
(*   IN THE INPUT STREAM, RETURNS THE NUMBER    *)
(*   OF DIGITS AND THE INTEGER VALUE            *)
(************************************************)

BEGIN

  DIGITS := DIGITS + 1 ;
  VALUE  := VALUE * BASE  +  CHARVAL(CH) ;
  INCHAR( CH ) ;
  WHILE  CH IN ['0' .. '9']  DO DIGITI( DIGITS, VALUE) ;

  IF  CH = '_'  THEN  BEGIN
    INCHAR( CH ) ;
    IF  CH IN ['0' .. '9']
      THEN  DIGITI( DIGITS, VALUE )
      ELSE  ERROR( 3, 1, 0 )   (* DIGIT MUST FOLLOW '_' *)
    END  (* IF *)

END ;  (*  DIGIT_INTEGER  *)
```

Figure A-5: Integer Digit Lexical Analyzer

B. COMPILER USER'S GUIDE

B.3 Introduction

This appendix describes the input accepted by the test compiler and the output which results. This guide is a continuation of the User's Guide prepared by Garlington [Ref 10]. Several example programs are included to demonstrate the various modes in which the compiler may be run.

Input: Input to the program is an Ada text file whose constructs have been included as part of the implemented subset. The language constructs used to compose input programs are listed below.

1. Integer variables. Number declarations and variable initializations are not implemented.

2. Package declarations.

3. Procedures and functions with parameters (mode types may be specified).

4. Task declarations.

5. Selected components may be used to open visibility to objects that are within scope but which are not directly visible.

6. Most integer arithmetic or Boolean expressions may be used including those using short circuit conditions. (REM, **, &, IN are not implemented)

7. The following statements may be used:

a. Assignment

b. Procedure, function or entry calls

c. Exit

d. Return

e. IF THEN ELSEIF ELSE

f. Accept

g. Loops (except FOR loop)

Output: The output of the program is dependent on a specially defined pragma. A pragma is the Ada construct which is used to convey information to the compiler. Pragmas serve to provide the translator with information that is relevant for the translation process, but which does not affect the semantics of the program [Ref 17:14-13]. The TOGGLE pragma was added to allow more direct control of the program throughout its development. Its format is:

PRAGMA TOGGLE (<OPTION_STRING>)

where <OPTION_STRING> is composed of selections from the following list of options: EXECUTE, TRACESTORE, TRACETOKEN, TRACEPARSE, PRINTCODE, PRINTTABLE, and PRINTSTACK. Multiple selections must be separated by commas.

All of these options are initially off. To select an option, list it in an option string, and the compiler's output will be as defined below:

EXECUTE; If no errors are detected in the input program, the program will be executed.

TRACESTORE: TRACESTORE will do nothing unless EXECUTE is also selected. If EXECUTE is selected, each value stored during execution of an ISTORE or ENTISTORE command will be printed.

TRACEPARSE: Each transition or reduction made by the parsing automation is printed. This listing is fairly long even for a short program.

TRACETOKEN: The representation of each token passed from the scanner to the parser is printed. This representation consists of the token's vocabulary index as output from the automatic parser generator.

PRINTCODE: The code generated by the compiler is formatted and printed.

PRINTTABLE: Prints the symbol table chain for each symbol table entry.

PRINTSTACK: The current values of the semantic stack are transferred to the file STACK.OUT.

B.4 Examples

The following examples illustrate the effects of selecting these options given a simple program as input.

```
ADA-P COMPILER                                   PAGE 1
AIR FORCE INSTITUTE OF TECHNOLOGY                15-Nov-81   69121283

   1 -- This example illustrates the compiler's output with
   2 -- none of the pragma toggles set ( no control information)
   3
   4 procedure MAIN is
   5   A : integer ;
   6 begin
   7   A := 3 ;
   8   PUT (" A = ");   PUT_LINE (A) ;
   9 end MAIN ;
```

Figure B-1: Compiler Execution - No Controls

```
ADA-P COMPILER                                   PAGE 1
AIR FORCE INSTITUTE OF TECHNOLOGY                15-Nov-81   69261433

   1 -- Now the same program is input to the compiler
   2 -- with the EXECUTION option set.
   3
   4 pragma TOGGLE (EXECUTE) ;
   5
   6 procedure MAIN is
   7   A : integer ;
   8 begin
   9   A := 3 ;
  10   PUT (" A = ");   PUT_LINE (A) ;
  11 end MAIN ;

*** PROGRAM EXECUTION ***

A =             3
```

Figure B-2: Compiler Execution - EXECUTE Option

```
ADA-P COMPILER                                    PAGE 1
AIR FORCE INSTITUTE OF TECHNOLOGY                 15-Nov-81  69409500

   1 -- Again, the same program is input to the compiler,
   2 -- but now the PRINTCODE option is selected.
   3
   4 pragma TOGGLE (PRINTCODE) ;
   5
   6 procedure MAIN is
   7   A : integer ;
   8 begin
   9   A := 3 ;
  10   PUT (" A = ");   PUT_LINE (A) ;
  11 end MAIN ;


*** PRAGMA PRINT_CODE ***

 INDEX  MNEMONIC   LEVEL  ADDRESS
    0   JMP          0        1
    1   INCT         0       17
    2   ILOADCONST   0        3
    3   ISTORE       0       16
    4   SPUT         0        5
    5   DATA         0       32
    6   DATA         0       65
    7   DATA         0       32
    8   DATA         0       61
    9   DATA         0       32
   10   ILOAD        0       16
   11   IPUT         1        0
   12   RETURN       0        0
```

Figure B-3: Compiler Execution - PRINTCODE Option

```
ADA-P COMPILER                                  PAGE 1
AIR FORCE INSTITUTE OF TECHNOLOGY               15-Nov-81   69740566

-- Now the TRACESTORE option is selected.
-- This option will print the value stored during
-- the execution of the ISTORE instruction.

pragma TOGGLE (EXECUTE,TRACESTORE) ;

procedure MAIN is
  A : integer ;
begin
  A := 3 ;
  PUT (" A = ");   PUT_LINE (A) ;
end MAIN ;


*** PROGRAM EXECUTION ***



*** PRAGMA TRACESTORE ***

EACH VALUE STORED DURING EXECUTION OF AN ISTORE COMMAND IS LISTED

           3
A =              3
```

Figure B-4: Compiler Execution - TRACESTORE Option

```
ADA-P COMPILER                                  PAGE 1
AIR FORCE INSTITUTE OF TECHNOLOGY               16-Nov-81  85211283

   1 -- Next the TRACETOKEN option is selected.
   2 -- This option prints the integer value of the next token.
   3 -- The integer value of the next token corresponds to the
   4 -- token's position in the list of terminals produced by
   5 -- the LRS Generator in LIST.LST.
   6
   7 pragma TOGGLE(TRACETOKEN) ;
   8
   9 procedure MAIN is
NEXTSYM =   73
NEXTSYM =    9
NEXTSYM =   59
  10   A : integer ;
NEXTSYM =    9
NEXTSYM =   19
NEXTSYM =    9
NEXTSYM =   21
  11 begin
NEXTSYM =   38
  12   A := 3 ;
NEXTSYM =    9
NEXTSYM =   20
NEXTSYM =   10
NEXTSYM =   21
  13   PUT (" A = ");   PUT_LINE (A) ;
NEXTSYM =    9
NEXTSYM =    4
NEXTSYM =   11
NEXTSYM =    5
NEXTSYM =   21
NEXTSYM =    9
NEXTSYM =    4
NEXTSYM =    9
NEXTSYM =    5
NEXTSYM =   21
  14 end MAIN ;
NEXTSYM =   49
NEXTSYM =    9
NEXTSYM =   21
NEXTSYM =   49
NEXTSYM =   49
```

Figure B-5: Compiler Execution - TRACETOKEN Option

```
ADA-P COMPILER                                  PAGE 1
AIR FORCE INSTITUTE OF TECHNOLOGY               15-Nov-81  71379383

    1 -- Now the lengthy output generated by the TRACEPARSE
    2 -- option is demonstrated. Each transition or reduction
    3 -- performed by the parsing automaton is printed.
    4 --
    5 -- Since the program listing is hidden among the trace
    6 -- of the parsing of the program we include in these
    7 -- comments the program which the compiler is parsing:
    8 --
    9 -- pragma TOGGLE (TRACEPARSE) ;
   10 --
   11 -- procedure MAIN is
   12 --    A : integer ;
   13 -- begin
   14 --    A := 3 ;
   15 --    PUT (" A = ");   PUT_LINE (A) ;
   16 -- end MAIN ;
   17 --
   18 --
   19
   20 pragma TOGGLE (TRACEPARSE) ;
PRODUCTION  18 AND TRANSITION FROM STATE   2 TO STATE   7
                   TRANSITION FROM STATE   7 TO STATE  41
   21
   22 procedure MAIN is
PRODUCTION  16 AND TRANSITION FROM STATE   2 TO STATE   8
PRODUCTION  15 AND TRANSITION FROM STATE   2 TO STATE   9
PRODUCTION 393 NOT IN SEMANTIC ****
PRODUCTION 393 AND TRANSITION FROM STATE   9 TO STATE  46
PRODUCTION 379 NOT IN SEMANTIC ****
PRODUCTION 379 AND TRANSITION FROM STATE   2 TO STATE   6
                   TRANSITION FROM STATE   6 TO STATE  18
PRODUCTION 278 AND TRANSITION FROM STATE   6 TO STATE  34
                   TRANSITION FROM STATE  34 TO STATE  11
PRODUCTION   8 AND TRANSITION FROM STATE  34 TO STATE 113
PRODUCTION 286 AND TRANSITION FROM STATE  34 TO STATE 112
PRODUCTION 280 AND TRANSITION FROM STATE   6 TO STATE  35
PRODUCTION 290 AND TRANSITION FROM STATE  35 TO STATE 116
PRODUCTION 288 AND TRANSITION FROM STATE 116 TO STATE 210
PRODUCTION 279 AND TRANSITION FROM STATE   6 TO STATE  37
PRODUCTION 275 AND TRANSITION FROM STATE   6 TO STATE  38
                   TRANSITION FROM STATE  38 TO STATE 117
   23   A : integer ;
PRODUCTION 281 AND TRANSITION FROM STATE   6 TO STATE  39
                   TRANSITION FROM STATE  39 TO STATE  11
```

```
PRODUCTION   8 AND TRANSITION FROM STATE  39 TO STATE  80
                   TRANSITION FROM STATE  80 TO STATE 186
PRODUCTION  48 AND TRANSITION FROM STATE 186 TO STATE 296
                   TRANSITION FROM STATE 296 TO STATE  11
PRODUCTION   8 AND TRANSITION FROM STATE 296 TO STATE 123
PRODUCTION 126 AND TRANSITION FROM STATE 296 TO STATE 456
PRODUCTION  60 AND TRANSITION FROM STATE 296 TO STATE 457
PRODUCTION  50 AND TRANSITION FROM STATE 457 TO STATE 584
PRODUCTION  42 AND TRANSITION FROM STATE  39 TO STATE  84
PRODUCTION  29 AND TRANSITION FROM STATE  39 TO STATE  74
                   TRANSITION FROM STATE  74 TO STATE 183
  24 begin
PRODUCTION  27 AND TRANSITION FROM STATE  39 TO STATE  73
PRODUCTION  26 AND TRANSITION FROM STATE  39 TO STATE 119
PRODUCTION 283 AND TRANSITION FROM STATE   6 TO STATE  40
                   TRANSITION FROM STATE  40 TO STATE 104
  25   A := 3 ;
PRODUCTION 209 AND TRANSITION FROM STATE 104 TO STATE 198
                   TRANSITION FROM STATE 198 TO STATE  11
PRODUCTION   8 AND TRANSITION FROM STATE 198 TO STATE 334
PRODUCTION 126 AND TRANSITION FROM STATE 198 TO STATE 342
                   TRANSITION FROM STATE 342 TO STATE 451
PRODUCTION  47 AND TRANSITION FROM STATE 342 TO STATE 500
                   TRANSITION FROM STATE 500 TO STATE 132
PRODUCTION  11 AND TRANSITION FROM STATE 500 TO STATE 152
PRODUCTION 142 AND TRANSITION FROM STATE 500 TO STATE 149
PRODUCTION 199 AND TRANSITION FROM STATE 500 TO STATE 155
PRODUCTION 197 AND TRANSITION FROM STATE 500 TO STATE 145
PRODUCTION 196 AND TRANSITION FROM STATE 500 TO STATE 159
PRODUCTION 190 AND TRANSITION FROM STATE 500 TO STATE 157
PRODUCTION 173 AND TRANSITION FROM STATE 500 TO STATE 156
PRODUCTION 160 AND TRANSITION FROM STATE 500 TO STATE 611
PRODUCTION 234 AND TRANSITION FROM STATE 198 TO STATE 326
PRODUCTION 215 NOT IN SEMANTIC ****
PRODUCTION 215 AND TRANSITION FROM STATE 198 TO STATE 347
PRODUCTION 213 AND TRANSITION FROM STATE 198 TO STATE 349
PRODUCTION 207 NOT IN SEMANTIC ****
PRODUCTION 207 AND TRANSITION FROM STATE 104 TO STATE 200
                   TRANSITION FROM STATE 200 TO STATE 350
  26   PUT (" A = ");   PUT_LINE (A) ;
PRODUCTION 205 AND TRANSITION FROM STATE 104 TO STATE 201
PRODUCTION 209 AND TRANSITION FROM STATE 201 TO STATE 198
                   TRANSITION FROM STATE 198 TO STATE  11
PRODUCTION   8 AND TRANSITION FROM STATE 198 TO STATE 334
PRODUCTION 126 AND TRANSITION FROM STATE 198 TO STATE 342
                   TRANSITION FROM STATE 342 TO STATE 214
PRODUCTION 150 AND TRANSITION FROM STATE 342 TO STATE 150
                   TRANSITION FROM STATE 150 TO STATE 111
```

```
PRODUCTION  10 AND TRANSITION FROM STATE 150 TO STATE 158
PRODUCTION 141 AND TRANSITION FROM STATE 150 TO STATE 149
PRODUCTION 199 AND TRANSITION FROM STATE 150 TO STATE 155
PRODUCTION 197 AND TRANSITION FROM STATE 150 TO STATE 145
PRODUCTION 196 AND TRANSITION FROM STATE 150 TO STATE 159
PRODUCTION 190 AND TRANSITION FROM STATE 150 TO STATE 241
PRODUCTION 173 AND TRANSITION FROM STATE 150 TO STATE 156
PRODUCTION 160 AND TRANSITION FROM STATE 150 TO STATE 236
PRODUCTION 151 NOT IN SEMANTIC ****
PRODUCTION 151 AND TRANSITION FROM STATE 150 TO STATE 238
PRODUCTION 148 AND TRANSITION FROM STATE 342 TO STATE 146
                   TRANSITION FROM STATE 146 TO STATE 229
PRODUCTION 147 AND TRANSITION FROM STATE 342 TO STATE 216
PRODUCTION 130 AND TRANSITION FROM STATE 198 TO STATE 124
PRODUCTION 127 AND TRANSITION FROM STATE 198 TO STATE 342
PRODUCTION 216 AND TRANSITION FROM STATE 198 TO STATE 347
PRODUCTION 213 AND TRANSITION FROM STATE 198 TO STATE 349
PRODUCTION 207 NOT IN SEMANTIC ****
PRODUCTION 207 AND TRANSITION FROM STATE 201 TO STATE 353
                   TRANSITION FROM STATE 353 TO STATE 526
PRODUCTION 206 AND TRANSITION FROM STATE 104 TO STATE 201
PRODUCTION 209 AND TRANSITION FROM STATE 201 TO STATE 198
                   TRANSITION FROM STATE 198 TO STATE  11
PRODUCTION   8 AND TRANSITION FROM STATE 198 TO STATE 334
PRODUCTION 126 AND TRANSITION FROM STATE 198 TO STATE 342
                   TRANSITION FROM STATE 342 TO STATE 214
PRODUCTION 150 AND TRANSITION FROM STATE 342 TO STATE 150
                   TRANSITION FROM STATE 150 TO STATE  11
PRODUCTION   8 AND TRANSITION FROM STATE 150 TO STATE 123
PRODUCTION 126 AND TRANSITION FROM STATE 150 TO STATE 239
PRODUCTION 200 AND TRANSITION FROM STATE 150 TO STATE 155
PRODUCTION 197 AND TRANSITION FROM STATE 150 TO STATE 145
PRODUCTION 196 AND TRANSITION FROM STATE 150 TO STATE 159
PRODUCTION 190 AND TRANSITION FROM STATE 150 TO STATE 241
PRODUCTION 173 AND TRANSITION FROM STATE 150 TO STATE 156
PRODUCTION 160 AND TRANSITION FROM STATE 150 TO STATE 236
PRODUCTION 151 NOT IN SEMANTIC ****
PRODUCTION 151 AND TRANSITION FROM STATE 150 TO STATE 238
PRODUCTION 148 AND TRANSITION FROM STATE 342 TO STATE 146
                   TRANSITION FROM STATE 146 TO STATE 229
PRODUCTION 147 AND TRANSITION FROM STATE 342 TO STATE 216
PRODUCTION 130 AND TRANSITION FROM STATE 198 TO STATE 124
PRODUCTION 127 AND TRANSITION FROM STATE 198 TO STATE 342
PRODUCTION 216 AND TRANSITION FROM STATE 198 TO STATE 347
PRODUCTION 213 AND TRANSITION FROM STATE 198 TO STATE 349
PRODUCTION 207 NOT IN SEMANTIC ****
PRODUCTION 207 AND TRANSITION FROM STATE 201 TO STATE 353
                   TRANSITION FROM STATE 353 TO STATE 526
```

```
  27 end MAIN ;
PRODUCTION 206 AND TRANSITION FROM STATE 104 TO STATE 201
PRODUCTION 263 AND TRANSITION FROM STATE 201 TO STATE 352
PRODUCTION 262 AND TRANSITION FROM STATE  40 TO STATE 120
                   TRANSITION FROM STATE 120 TO STATE 212
                   TRANSITION FROM STATE 212 TO STATE  11
PRODUCTION   8 AND TRANSITION FROM STATE 212 TO STATE 113
PRODUCTION 286 AND TRANSITION FROM STATE 212 TO STATE 360
PRODUCTION 285 AND TRANSITION FROM STATE 212 TO STATE 361
PRODUCTION 282 AND TRANSITION FROM STATE   6 TO STATE  36
PRODUCTION 386 AND TRANSITION FROM STATE   6 TO STATE  20
PRODUCTION 388 AND TRANSITION FROM STATE   6 TO STATE  24
PRODUCTION 380 AND TRANSITION FROM STATE   2 TO STATE   4
                   TRANSITION FROM STATE   4 TO STATE  13
PRODUCTION   4 AND TRANSITION FROM STATE   2 TO STATE   5
PRODUCTION   3 AND TRANSITION FROM STATE   2 TO STATE  10
                   TRANSITION FROM STATE  10 TO STATE  48
```

Figure B-6: Compiler Execution - TRACEPARSE Option

C. CREATING THE COMPILER

C.5 Introduction

The Ada compiler was created through the use of several parts. The Ada grammar was processed by the Lawrence Livermore Laboratories (LLL) Language Processor. The parser tables generated by LLL's LR Parser were translated to produce initialization values in PASCAL source code. These PASCAL statements were merged with the PASCAL program ADAPAR.TRF to produce the complete Ada compiler ADAPAR.PAS.

C.6 LR Parser Generator

The Lawrence Livermore Laboratories (LLL) Language Processor has an automatic parser generator. "The parser generator reads a context-free grammar in a modified BNF format and produces tables to construct a parse of an input stream supplied by a (locally written) lexical analyzer" [Ref 24].

The LLL's LR Parser Generator expects the grammar in an input file called GRAM.BNF and produces the files LIST.LPT and TABLES.OUT as output. The LR Parser Generator will ask for the input grammar, if file GRAM.BNF does not exist. The LR Parser Generator was maintained in FORTRAN source code as file LRS.FOR, and its executable form as file LRS.EXE. Figure C-1 is an example of the execution of the LR Parser

Generator. The Ada grammar used for this compiler was originally produced by Intermetrics [Ref 18].

```
.RUN LRS.EXE

END OF EXECUTION
CPU TIME: 7:29.77        ELAPSED TIME: 17:34.33
EXIT

.
```

Figure C-1: Example LRS.EXE Execution

LIST.LPT is a listing of the grammar terminals, non-terminals, vocabulary cross-references, productions, and configuration sets with the possible transitions or reductions. LIST.LPT will also list any errors in the grammar. TABLES.OUT contains the integer values for the parser generator tables which must be translated before inclusion in the compiler.

C.7 Translation

The parser tables generated by the LR Parser Generator, TABLES.OUT, were modified by adding at the beginning of the file the position of the first reserved word in the grammar and the position of the last reserved word in the grammar which was input to LRS.EXE, currently 31 and 92 respectively. This modified file, designated TABLES.NEW, was then run through the translator program, TRANS.PAS, which translates the tables of integers into PASCAL initialization values for the parser tables. The output from the translator program was named TABLES.PAS. Following the table values are the constant values needed by TRANS.TRF, the compiler. Figure C-2 is an example of the execution of the translator program.

```
.EXECUTE TRANS.PAS
PASCAL: TRANSL
LINK:   Loading
[LNKXCT TRANSL Execution]
INPUT       : TABLES.NEW
OUTPUT      : TABLES.PAS

EXIT

.
```

Figure C-2: Example TRANS.PAS Execution

C.8 Merging

The parser table initialization values were merged into the Ada compiler ADAPAR.TRF. This was accomplished by dividing the compiler, ADAPAR.TRF, into two files at the point where the initialization of the parser tables is indicated. The constants at the end of TABLES.PAS were removed, modifying the value of any constant in the compiler which required change. Due to the restriction on the size of INITPROCEDURE, the parser table initialization values must be divided into several INITPROCEDUREs. The current breakpoints for INITPROCEDURE are before FTRN, TRAN [1246], TRAN [3246], LEN, and LS[1100]. PIP was used to merge the three files into one; the first part of ADAPAR.TRF, TABLES.PAS, and the second part of ADAPAR.TRF producing ADAPAR.PAS. ADAPAR.PAS is the final PASCAL source code of the Ada compiler.

C.9 Note

All references to files refer to files on the AFAL DEC-10, WPAFB. The filenames were maintained under PPN[6664,410]. Filenames which include PAR, such as ADAPAR, may be the original file as produced by Garlington. Filenames which include PAT, such as ADAPAT, are those produced by Werner.

D. TESTS

Introduction to tests.

D.10 Tests

The Ada Compiler Validation (ACVC) tests were acquired through Dr John B. Goodenough of Intermetrics via the ARPAnet File Transfer Program (FTP) from ARPAnet host USC-ECLB. The test programs are currently archived on AFAL DEC-10 on tape titles ACVC_TESTS reelid # 492 under programmer project number (PPN) [6664,410].

D.11 Examples

The following examples demonstrate the use of the ACVC tests on the AFIT-Ada compiler.

```
-- Test AFIT-Ada reserve words

procedure MAIN is
  RETURN : integer ;
begin
  RETURN := 3 ;
  PUT (" RETURN = ");   PUT_LINE (RETURN) ;
end MAIN ;
```

Figure D-1: Test Reserve Word

```
ADA-P COMPILER                                         PAGE 1
AIR FORCE INSTITUTE OF TECHNOLOGY                      24-Nov-81

   1 -- Test AFIT-Ada reserve words
   2
   3 procedure MAIN is
   4   RETURN : integer ;
****        © 5 *IDENTIFIER* BEGIN FOR FUNCTION GENERIC NEW
              PACKAGE PRAGMA PROCEDURE SUBTYPE TASK TYPE USE
   5 begin
   6   RETURN := 3 ;
   7   PUT (" RETURN = ");   PUT_LINE (RETURN) ;
   8 end MAIN ;

                         PROGRAM ERRORS
                         **************

   5 : IMPROPER CONSTRUCTION, EXPECTED SYMBOL LIST FOLLOWS
```

Figure D-2: Test Reserve Word Output

```
-- INTERMETRICS TEST MODIFIED TO TEST AFIT-ADA COMPILER

-- THOSE PORTIONS OF THE INTERMETRICS TEST NOT CURRENTLY
-- TESTABLE ON AFIT-ADA COMPILER ARE TRANSFORMED TO COMMENTS

-- 21 NOVEMBER 1981

-- A22002A.ADA

-- CHECK DELIMITERS (INCLUDING COMPOUND SYMBOLS) ARE ACCEPTED.
-- & ' ( ) * + , - . / : ; < = > |
-- => .. ** := /= >= <= << >> <>

-- DCB 1/5/80
-- DCB 1/22/80
-- JRK 10/21/80

--WITH REPORT;
PROCEDURE A22002A IS

--      USE REPORT;

     C1:STRING(1..6);      -- USE OF : ( ) ..
     C2:STRING(1..6);
     C3:STRING(1..12);

     I1, I2, I3, I4 : INTEGER;

     TYPE TABLE IS ARRAY (1..10) OF INTEGER;
--   A : TABLE := (2|4|10=>1,1|3|5..9=>0);  -- USE OF | => ,

     TYPE BUFFER IS
          RECORD
               LENGTH : INTEGER;
               POS    : INTEGER;
               IMAGE  : INTEGER;
          END RECORD;
--      R1 : BUFFER;

     TYPE BOX IS ARRAY(INTEGER RANGE<>) OF INTEGER; -- USE OF <>
```

```
BEGIN
--     TEST ("A22002A", "CHECK THAT DELIMITERS ARE ACCEPTED");

     C1 := "ABCDEF";
     C2 := "GHIJKL";
     C3 := C2&C1;          -- USE OF &

     I1:=2*(3-1+2)/2;I2:=2**3;  -- USE OF := ( ) * + - / ; **

     I3 := A'LAST;         -- USE OF '

     R1.POS := 3;          -- USE OF .

     <<LABEL>>  -- USE OF << >>

     IF I1>2 AND
        I1=4 AND
        I1<8 AND
        I2=8 AND
        I2/=3 AND
        I3<=10 AND
        I3>=10 THEN        -- USE OF < = > /= <= >=
          NULL;
     END IF;

--     RESULT;
END A22002A;
```

Figure D-3: ACVC Test A22002A - Tests Delimiters

```
ADA-P COMPILER                                       PAGE 1
AIR FORCE INSTITUTE OF TECHNOLOGY                    24-Nov-81

-- INTERMETRICS TEST MODIFIED TO TEST AFIT-ADA COMPILER

-- THOSE PORTIONS OF THE INTERMETRICS TEST NOT CURRENTLY
-- TESTABLE ON AFIT-ADA COMPILER ARE TRANSFORMED TO COMMENTS

-- 21 NOVEMBER 1981

-- A22002A.ADA

-- CHECK DELIMITERS (INCLUDING COMPOUND SYMBOLS) ARE ACCEPTED.
-- & ' ( ) * + , - . / : ; < = > |
-- => .. ** := /= >= <= << >> <>

-- DCB 1/5/80
-- DCB 1/22/80
-- JRK 10/21/80

--WITH REPORT;
PROCEDURE A22002A IS

--      USE REPORT;

     C1:STRING(1..6);       -- USE OF : ( ) ..
     C2:STRING(1..6);
     C3:STRING(1..12);

     I1, I2, I3, I4 : INTEGER;

     TYPE TABLE IS ARRAY (1..10) OF INTEGER;
--   A : TABLE := (2|4|10=>1,1|3|5..9=>0);  -- USE OF | => ,

     TYPE BUFFER IS
          RECORD
               LENGTH : INTEGER;
               POS    : INTEGER;
               IMAGE  : INTEGER;
          END RECORD;
--      R1 : BUFFER;

     TYPE BOX IS ARRAY(INTEGER RANGE<>) OF INTEGER; -- USE OF<>
```

```
  41
  42 BEGIN
  43 --      TEST ("A22002A", "CHECK THAT DELIMITERS ARE ACCEPTED");
  44
  45      C1 := "ABCDEF";
****                      ©25
  46      C2 := "GHIJKL";
  47      C3 := C2&C1;         -- USE OF &
  48
  49      I1:=2*(3-1+2)/2;I2:=2**3;  -- USE OF := ( ) * + - / ; **
  50
  51      I3 := A'LAST;        -- USE OF '
  52
  53      R1.POS := 3;         -- USE OF .
  54
  55      <<LABEL>>  -- USE OF << >>
  56
  57      IF I1>2 AND
  58         I1=4 AND
  59         I1<8 AND
  60         I2=8 AND
  61         I2/=3 AND
  62         I3<=10 AND
  63         I3>=10 THEN       -- USE OF < = > /= <= >=
  64           NULL;
  65      END IF;
  66
  67 --      RESULT;
  68 END A22002A;
```

```
                    PROGRAM ERRORS
                    **************

  25 : CURRENT IMPLEMENTATION ALLOWS ONLY INTEGER EXPRESSIONS
```

Figure D-4: ACVC Test A2202A - Output

E. TERMINOLOGY

ALPHABET	- a finite set of tokens of which sentences in the language are composed [Ref 3:15].
CANONICAL	- in simplest or standard form [Ref 20:217].
GRAMMAR	- a set of structural rules defining the legal contexts of tokens in sentences [Ref 3:15].
LEXICAL	- pertaining to the words or vocabulary of a language as contrasted with its grammatical and syntactical aspects [Ref 20:825].
LOOKAHEAD	- the next character or symbol past the current state
NONTERMINAL	- the special symbols that denote sets of strings [Ref 1:127], some set of strings in the alphabet which is not itself in the alphabet [Ref 3:18], e.g. statement-list, expression, term.
PRODUCTION	- (rewriting rules) define the ways in which the nonterminals may be built up from one another and from the terminals [Ref 1:127]; has the general form X ::= Y where X and Y are strings in the terminal and nonterminal sets of a given grammar, the string Y may be the empty string but X cannot be [Ref 3:19].
REDUCTION	- the replacement of the right side of a production by the left side of the production [Ref 1:150].
SEMANTICS	- a set of rules that define the operational effect of any program written in the language when translated and executed on some machine [Ref 3:15], the rules that give meaning to programs [Ref 1:28].
SYNTAX	- the set of rules which determine whether or not a string is a valid, or well-formed, program [Ref 1:28].

TERMINAL	- the basic symbols of which strings in the language are composed, with reference to programming languages the word "token" is a synonym for "terminal" [Ref 1:127].
TOKEN	- a character or sequence of characters whose significance is processed collectively rather than individually [Ref 1:33], usually dealt with in two classes: specific strings, i.e. reserved words, and classes of strings, i.e. identifiers, constants, or labels [Ref 1:10].
TRANSITION	- movement or passage from on state to another [Ref 20:1505].

VITA

Patrick Reemelin Werner was born on 1 April 1950 at Carswell AFB in Forth Worth, Texas, to Crowell B Werner and Mary Katherine (Reemelin) Werner. In 1968, he graduated from Highland High School in Albuquerque, New Mexico. Matriculating at Texas Technological College in Lubbock, Texas, he completed his undergraduate education at the University of California in Berkeley, California. Since receiving his Bachelor of Science in Electrical Engineering and Computer Science in June 1973, he has completed graduate courses at the University of Southern California and the University of Oklahoma. Commissioned through the AFROTC program, his first assignment was with the Engineering Division, Oklahoma City Air Logistics Center at Tinker AFB, Oklahoma. There he worked on the B-52, C-135, and A-7 aircraft, the E-3A Airborne Warning and Control System (AWACS), and various automatic test equipment. In 1977 he was assigned to the Directorate of Engineering Services, Air Force Acquisition Logistics Division (AFALD), at Wright-Patterson AFB, Ohio. Here he worked to assure logistics supportability of computer systems under acquisition and was Project Manager for the Engineering Data Requisition and Index System. In June 1980, he entered the School of Engineering, Air Force Institute of Technology.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/GCS/MA/81D-7	2. GOVT ACCESSION NO. AD-A115	3. RECIPIENT'S CATALOG NUMBER 479
4. TITLE (and Subtitle) TOWARD ADA: CONTINUING DEVELOPMENT OF AN ADA COMPILER		5. TYPE OF REPORT & PERIOD COVERED MASTERS THESIS
		6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Patrick R. Werner Capt USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT/EN) Wright-Patterson AFB, OH 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Avionics Laboratory (AFWAL/AAAF-2) Wright-Patterson AFB, OH 45433		12. REPORT DATE December 1981
		13. NUMBER OF PAGES 72
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) Unclassified
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

15 APR 1982

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

[illegible]

Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB, OH 45433

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

~~F. C. Lynch, Major,~~ USAF
Di~~rector of Information~~on

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

ADA
COMPILER
LR GRAMMAR
PARSE

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This thesis involved the continuing development of the AFIT-Ada compiler. Basic concepts of the compilation process were reviewed. The design and structure of the AFIT-Ada compiler were examined. Tests from the Ada Compiler Capability system were applied to the compiler. Sections of the compiler were rewritten in a more structured manner. This development was performed on the DEC-10 system of the Air Force Avionics Laboratory at Wright-Patterson AFB.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

MED

8